Investigation of Scratchpad Memory for Preemptive Multitasking

Jack Whitham, Robert I. Davis and Neil C. Audsley
Real-time Systems Group
University of York, York, UK
jack.whitham/rob.davis/neil.audsley@york.ac.uk

Sebastian Altmeyer
Compiler Design Lab
Saarland University, Germany
altmeyer@cs.uni-saarland.de

Claire Maiza
Verimag
INP Grenoble, France
claire.maiza@imag.fr

***Abstract*—We present a multitasking scratchpad memory reuse scheme (MSRS) for the dynamic partitioning of scratchpad memory between tasks in a preemptive multitasking system. We specify a means to compute the worst-case response time (WCRT) and schedulability of task sets executed using MSRS. Our scratchpad-related preemption delay (SRPD) is an analog of cache-related preemption delay (CRPD), proposed in previous work as a way to compute the worst-case cost imposed upon a preempted task by preemption in a multitasking system. Unlike CRPD, however, SRPD is independent of the number of tasks and the local memory size.**

We compare SRPD with CRPD by experiment and determine that neither dominates the other, i.e. either may be better for certain task sets. However, MSRS leads to improved schedulability versus cache when contention for local memory space is high, either because the local memory size is small, or because the task set is large, provided that the cost of loading blocks from external memory to scratchpad is similar to the cost of loading blocks into cache.

I. Introduction

Scratchpad memory (SPM) is a form of *local memory*. Local memory is the memory resource closest to the CPU. Typically a cache is used [1], but SPM can be substituted, bringing various advantages for time-predictable operation [2] and reduced energy usage [3]. SPM is present in a number of CPUs including Cell [4].

There has been recent interest in using SPM within real-time systems instead of cache [5], [6], [7], [8]. But support for preemptive multitasking is a problem: how should the SPM be managed if tasks can preempt each other? The SPM can be *statically partitioned*, allocating a fixed amount of space to every task [9]. However, large tasks (and large task sets) call for a *dynamic* scheme that allows the space used by one task to be reused by another [10]. We call this a *multitasking scratchpad reuse scheme* (MSRS).

The basic MSRS approach involves a new parameter for each task τ_j which specifies the quantity of SPM space required by τ_j. The OS reserves this space before τ_j begins, and restores the previous usage of the space when τ_j completes. MSRS allows this quantity to be anything from zero to the total number of blocks provided by the physical hardware [10]. The impact of a preempting task τ_j on a preempted task τ_i is limited to the time cost of reserving and restoring the SPM space required by τ_j. In this paper, this time cost is known as *scratchpad-related preemption delay* (SRPD).

With cache, space used by one task can be reused by another as a consequence of the cache replacement policy. In a hard real-time system it is important to establish the impact of a preempting task τ_j upon a preempted task τ_i. This impact is known as *cache-related preemption delay* (CRPD). CRPD may be determined by considering which cache blocks are reused by preempted tasks *and* evicted by preempting tasks. Recent work has established a number of approaches for this [11], [12], [13], [14], [15], [16], [17].

Figure 1 shows how SRPD and CRPD are compared in this paper. SRPD and CRPD consider the interactions between pairs of tasks τ_i and τ_j. In order to determine how SRPD/CRPD affect the schedulability of the system as a whole, we use *response time analysis* (RTA). RTA determines the time interval between the release of a task and its completion, which may include preemption delay. We use *worst-case response time* (WCRT) analysis [18] as this determines an upper bound on the response time, and hence indicates if task deadlines may not be met. To distinguish between SRPD/CRPD analysis alone, and SRPD/CRPD used in conjunction with WCRT analysis, we denote the latter by SRPD-RTA and CRPD-RTA respectively.

An MSRS named Carousel has been specified in an earlier paper and a basic comparison with cache has already been carried out [10]. However, this was a proof of concept. It made use of measurements instead of static analysis to determine *worst-case execution times* (WCETs). It used simulation rather than obtaining timings from real hardware, and it failed to properly distinguish between a specific implementation (Carousel) and the general properties of MSRS. Some of the improvements observed in [10] are specific to the (simulated) hardware considered, and do not generalize.

The contributions of this paper are as follows. We describe the MSRS approach in its simplest form. We specify SRPD as a way to determine the impact of task preemption with MSRS, and SRPD-RTA as a way to analyze the schedulability of systems using MSRS, maintaining an analogy to CRPD and CRPD-RTA throughout. Then, we use a working system implementation, based on Carousel but greatly simplified, to obtain the parameter values needed to compare the schedulability of task sets with both SRPD-RTA and CRPD-RTA as shown in Figure 1. We show that some task sets are schedulable with MSRS but not with cache and vice versa.

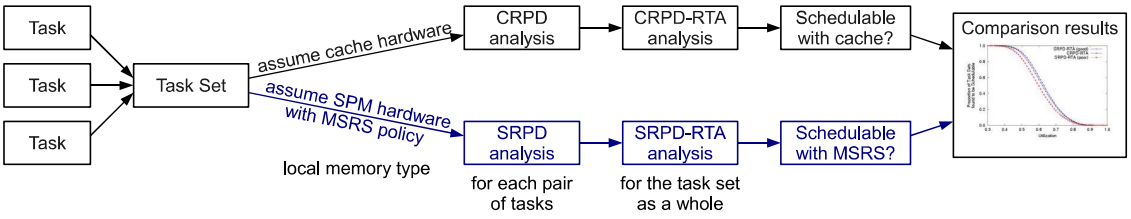

Fig. 1. How SRPD and CRPD are compared in this paper. Two levels of analysis are used. Firstly, SRPD/CRPD analysis determines the impact of each task τ_j on a lower-priority task τ_i. Secondly, SRPD-RTA/CRPD-RTA analysis determines whether a task set is schedulable given the overheads of SRPD/CRPD. CRPD and CRPD-RTA analysis are described in previous work (section II).

We show that MSRS is preferable when contention for local memory space is high, because the local memory size is small, or because the task set is large.

Our comparisons focus on simple cases that allow us to characterize the inherent differences between MSRS and cache and the benefits of one or the other. We use small benchmark tasks executed on a timing-compositional architecture [19] (entirely free of timing anomalies) with a perfect data cache, and either a direct-mapped instruction cache or an instruction SPM. The tasks are sufficiently small that each task fits entirely in local memory, but when combined into a task set, the total size is larger than local memory, and hence some reuse of local memory is required. This assumption means that no *conflict misses* occur [20], and furthermore, we can use a simple scheme to allocate SPM space to tasks, thus clarifying the comparison results.

The structure of this paper is as follows. Section II describes how CRPD-RTA analysis is carried out. Section III specifies SRPD-RTA analysis. Section IV obtains realistic parameter values for SRPD and CRPD from a working implementation, then carries out a comparison using the approach shown in Figure 1. Section V describes further comparisons of a similar nature, in which the reasons for the different behavior of MSRS and cache are explored, Section VI discusses related work and Section VII summarizes the findings.

II. Worst-Case Response Time

A real-time task set is *schedulable* if the *worst-case response time* (WCRT) R_i of each task τ_i is no greater than its *deadline* D_i. A *schedulability test* may be performed to determine if a particular task set is schedulable. We assume a set of n tasks scheduled using fixed priority preemptive scheduling. Each task τ_i has a unique priority i, with τ_1 having the highest priority, and task τ_n having the lowest priority.

A. WCRT General Form

The WCRT equation gives the maximum amount of time R_i between the *release* of a task τ_i (the point in time when τ_i becomes runnable) and the completion of τ_i. τ_i has a *period* T_i which is the minimum interval between invocations of τ_i, a bounded WCET C_i and a *blocking time* B_i which is the maximum amount of time τ_i can spend waiting for a shared resource due to the execution of lower priority tasks.

Let $\mathrm{hp}(i)$ be the set of tasks with a higher priority than τ_i. If $\mathrm{hp}(i)$ is not empty, then execution of τ_i may be suspended whilst $\tau_j \in \mathrm{hp}(i)$ is executed - this is called *interference*. Response time analysis [18] for constrained deadline tasks (where $D_i \leq T_i$) takes this time into account:

$$R_i = \max(B_i, CS^{\text{from}}) + CS^{\text{to}} + C_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^{\text{to}} + C_j + CS^{\text{from}}) \quad (1)$$

Equation (1) extends the standard response time analysis to include OS overheads relating to context switching and scheduling. CS^{to} is the time taken to switch to a task after an event releases it, and CS^{from} is the time taken to switch away from a task. Both are critical sections. Typically, CS^{to} and CS^{from} are implicitly considered as part of the execution time bound C_i; however, we need to handle these costs explicitly.

Although the context switch away from a job of task τ_i does not count directly towards its response time, which is measured up to the point at which the job completes its normal execution, this context switch may cause a delay to the following job of τ_i. For tasks with constrained deadlines ($D_i \leq T_i$), the maximum amount of additional interference or indirect blocking a previous job of task τ_i can cause to the next job of task τ_i is limited to the amount of *post-completion* execution carried out after the end of the task's normal execution, i.e. CS^{from}. Furthermore, this effect can only occur if the CPU remains busy with the context switch or higher-priority execution until after the next job of task τ_i is released. Hence jobs of task τ_i can suffer either blocking from lower-priority tasks, or interference due to post-completion execution related to the previous job of the same task, but not both. The effect can be accounted for by including CS^{from} in the blocking term $\max(B_i, CS^{\text{from}})$ in a similar way to the sufficient schedulability test given in Section 3.4 of [21].

The WCRT of task τ_i appears on both the left and right hand sides (RHS) of (1); however, as the RHS is a monotonically non-decreasing function of R_i, the equation can be solved via fixed point iteration. This starts with an initial value e.g. $R_i = C_i$ and iterates until R_i either converges or exceeds the task's deadline. We note that convergence can be speeded up by starting with a lower bound on R_i as an initial value [22].

The WCET of task τ_i depends on the local memory type. We use C_i^{cache} to mean the WCET assuming cache and

C_i^{spm} to mean the WCET assuming SPM, similarly for the blocking times B_i^{cache} and B_i^{spm}. C_i^{cache} and C_i^{spm} include the cost of executing instructions, and the cost of loading those instructions from external memory. C_i^{cache} and C_i^{spm} assume that τ_i is executed non-preemptively. Preemption delays are considered separately, as discussed in the following sections. B_i^{cache} and B_i^{spm} include blocking due to lower-priority tasks, and any post-completion execution related to task τ_i, specifically CS^{from}, and for SPM, the additional cost of restoring the SPM contents. B_i^{cache} and B_i^{spm} are formally defined later.

B. CRPD-RTA

When a task τ_i is preempted by task τ_j, some of the cache blocks used by τ_i may be evicted by τ_j. This means that the execution time of task τ_i is increased. Some additional cache misses occur as the evicted blocks are reloaded. The increase in execution time is the *cache-related preemption delay* (CRPD). CRPD can be integrated into WCRT analysis as described in [16], [17]. In this paper, the combination is called CRPD-RTA analysis.

CRPD can be applied to both instruction and data caches. Our focus in this paper is on direct-mapped instruction caches only, as these have all of the properties necessary for our investigation. In this paper we determine the CRPD using the ECB-Union/UCB-Union approach [16], which safely estimates the worst-case CRPD when task τ_j preempts τ_i.

For each task τ_i, we use WCET analysis to determine ECB_i, the set of *evicting cache blocks*. This set contains the cache blocks which *might* be used by τ_i as it executes [12]. ECB_i is determined by considering the addresses of the instructions in the task and where they map to within the cache.

We also use WCET analysis to determine UCB_i, the set of *definitely-cached useful cache blocks* (DC-UCBs) at each point in task τ_i, being the set of cache blocks that *must* be in cache at that point and may be reused *provided that* there is no preemption [15]. There is one DC-UCB set for each possible preemption point, but as a safe approximation, we can use whichever of these maximizes the subsequent equations.

The intersection between ECB sets and UCB sets for pairs of tasks is the matter of interest, because any cache block evicted by a higher-priority task τ_j *and* certainly used by a lower-priority task τ_i will result in a cache miss that is not included in C_i^{cache}. As shown in [12], the extra cache misses are taken into account within the CRPD-RTA equation by introducing $\gamma_{i,j}^{\text{CRPD}}$, the *preemption-related delay* imposed upon τ_i as a result of cache misses caused by τ_j. Note that τ_j may impose a preemption-related delay on τ_i by causing cache misses within τ_i itself and within tasks of intermediate priority between i and j. The CRPD-RTA equation is:

$$R_i^{\text{CRPD}} = B_i^{\text{cache}} + CS^{\text{to}} + C_i^{\text{cache}} + \sum_{j \in \text{hp}(i)} \left\lceil \frac{R_i^{\text{CRPD}}}{T_j} \right\rceil (CS^{\text{to}} + C_j^{\text{cache}} + CS^{\text{from}} + \gamma_{i,j}^{\text{CRPD}}) \quad (2)$$

where

$$B_i^{\text{cache}} = \max(B_i, CS^{\text{from}}) \quad (3)$$

C. UCB-Union

UCB-Union is one definition for $\gamma_{i,j}$. Introduced by Tan and Mooney [13], UCB-Union dominates an earlier ECB-only approach described by Busquets et al. [12].

Define $\text{aff}(i,j)$ as the set of tasks that might be preempted by τ_j whilst τ_i is running or preempted. This includes τ_i, and all tasks with a priority between i and j. Define $\text{BRT}^{\text{cache}}$ as the time needed to load a single cache block from external memory.

UCB-Union takes the union of all UCBs that might be preempted by τ_j whilst τ_i is running or preempted. It then excludes UCBs that are not evicted by τ_j. This gives an upper bound on the CRPD imposed by τ_j on τ_i [16]:

$$\gamma_{i,j}^{\text{ucbu}} = \text{BRT}^{\text{cache}} \left| \left(\bigcup_{\forall k \in \text{aff}(i,j)} \text{UCB}_k \right) \cap \text{ECB}_j \right| \quad (4)$$

D. ECB-Union

UCB-Union is safe, but it is not necessarily precise, and an alternative view of the problem can sometimes produce tighter bounds on $\gamma_{i,j}$.

The ECB-Union version of $\gamma_{i,j}$ finds the maximum number of UCBs belonging to some task in $\text{aff}(i,j)$ that could be evicted by τ_j including cases where τ_j is also preempted by some higher priority task in $\text{hp}(j)$. The definition is [16]:

$$\gamma_{i,j}^{\text{ecbu}} = \text{BRT}^{\text{cache}} \max_{\forall k \in \text{aff}(i,j)} \left\{ \left| \text{UCB}_k \cap \left(\bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h \right) \right| \right\} \quad (5)$$

Like UCB-Union, ECB-Union is safe but not necessarily precise. It is incomparable with UCB-Union, and in [16], Altmeyer, Davis and Maiza state that WCRT analysis should be performed for each task τ_i using both $\gamma_{i,j}^{\text{ucbu}}$ and $\gamma_{i,j}^{\text{ecbu}}$, with the minimum value of R_i used in each case:

$$R_i^{\text{CRPD}} = \min \left(R_i^{\text{ecbu}}, R_i^{\text{ucbu}} \right) \quad (6)$$

This is an important way to improve precision without compromising safety since neither UCB-Union nor ECB-Union may produce an exact WCRT.

III. Scratchpad-Related Preemption Delay (SRPD)

An instruction cache may be replaced by an SPM [5]. This substitution simplifies the CPU hardware, reduces its energy consumption [3] and may improve the precision of WCET analysis [2]. However, tasks must manage the local memory resource explicitly. Small tasks (and small numbers of tasks) may permit a static allocation of code to SPM, unchanged during execution, but all other cases force reuse of SPM space while tasks execute.

A. Multitasking Scratchpad Reuse Scheme

A *multitasking scratchpad reuse scheme* (MSRS) shares SPM space between multiple tasks.

MSRS involves extra steps during context switches. When the OS switches from one task (τ_i) to another (τ_j), the MSRS

ensures that all of the SPM space required by task τ_j is available for use by τ_j. Let S_j^{spm} be the total number of SPM blocks required by τ_j.

The basic MSRS implementation involves the following four steps, which take place whenever task τ_j executes:

Save
: The current contents of S_j^{spm} SPM blocks are saved in external memory. These belong to preempted tasks. The WCET of the *Save* step is C_j^{save}. If the blocks are read-only (e.g. code blocks) it is sufficient to only *Save* the address of each block.

Load
: A part of task τ_j is loaded from external memory to SPM. The WCET of all *Load* steps during τ_j is C_j^{load}. Up to S_j^{spm} blocks may be *Loaded* during each step.

Execute
: A part of task τ_j is executed from SPM. The WCET of all *Execute* steps during τ_j is C_j^{execute}.

Restore
: Upon completion, the original contents of the S_j^{spm} SPM blocks are restored. The WCET of the *Restore* step is C_j^{restore}.

C_j^{spm}, the WCET of task τ_j, may be defined as follows:

$$C_j^{\text{spm}} = C_j^{\text{load}} + C_j^{\text{execute}} \tag{7}$$

The CRPD analysis is not directly applicable to SPM because local memory blocks are not loaded implicitly as for cache. Instead, they are loaded explicitly during the *Load* and *Restore* steps. Nevertheless, the time cost of *Save* and *Restore* acts analogously to CRPD, increasing the WCRT of lower-priority tasks. Hence, we may define *scratchpad-related preemption delay* (SRPD) in terms of *Save* and *Restore*.

B. SRPD-RTA Equation

Scratchpad-related preemption delay (SRPD) defines γ^{SRPD} as follows:

$$\gamma_{i,j}^{\text{SRPD}} = C_j^{\text{save}} + C_j^{\text{restore}} \tag{8}$$

Like γ^{ucbu}, this is the increase in R_i for every case where τ_j may preempt τ_i: the time cost of *Saving* S_j^{spm} blocks and then *Restoring* them before returning to τ_i.

$\gamma_{i,j}^{\text{SRPD}}$ has no dependence on i. SRPD is independent of other tasks. The parameter i is included so that $\gamma_{i,j}^{\text{SRPD}}$ can be substituted into (2) as follows, giving the equation for SRPD-RTA:

$$R_i^{\text{SRPD}} = B_i^{\text{spm}} + CS^{\text{to}} + C_i^{\text{save}} + C_i^{\text{spm}} + \sum_{j \in \text{hp}(i)} \left\lceil \frac{R_i^{\text{SRPD}}}{T_j} \right\rceil (CS^{\text{to}} + C_j^{\text{spm}} + CS^{\text{from}} + \gamma_{i,j}^{\text{SRPD}}) \tag{9}$$

where

$$B_i^{\text{spm}} = \max(B_i, C_i^{\text{restore}} + CS^{\text{from}}) \tag{10}$$

We return to the contribution to B_i from lower priority tasks in Section III-D.

C. Regions

It is important to minimize (8), which means minimizing S_i^{spm}, the total number of SPM blocks used by task τ_i. The techniques required are common to all SPM allocation algorithms, because these involve mapping a program of size s into a memory of size k, where $s > k$ [3], [5].

The basic approach is to form *regions* to multiplex the SPM usage of τ_i so that multiple parts of τ_i reuse the same SPM space [5], reducing the total number of physical blocks required by τ_i. The same total number of blocks must still be loaded from external memory, as the combined size of all regions is the same as the task size. However, the storage space required for those blocks is only the size of the largest region, which is smaller than the size of the task.

The sizes of the m regions used by τ_i are $S_{i,1}^{\text{spm}}, S_{i,2}^{\text{spm}}, ..., S_{i,m}^{\text{spm}}$, and the loading cost for region x is $C_{i,x}^{\text{load}}$. The largest region size is S_i^{spm}:

$$S_i^{\text{spm}} = \max_{x \in [1,m]} S_{i,x}^{\text{spm}} \tag{11}$$

The region-forming model used in this paper assumes that τ_i is a linear sequence of regions, i.e. region $x+1$ always follows region x. For example, the parts of the task dedicated to initializing and completing some work may be placed in different regions to the main part of the task which actually carries out the work. The initialization and completion code is only executed once and can be discarded after use. More complex approaches to SPM allocation are possible, but the simple approach is effective when S_i^{spm} is no larger than the available SPM space. The region formation process adds a small amount of additional code to each region $x < m$ in order to trigger the *Load* for region $x+1$.

D. Blocking Time

B_i^{cache} and B_i^{spm} include the maximum amount of time τ_i may be forced to wait for a shared resource in use by a lower-priority task. In this paper, we assume tasks are independent, so blocking is due only to OS critical sections (CS^{to}, CS^{from}) and MSRS steps *Save*, *Load* and *Restore*. Cache is only affected by the OS:

$$B_i^{\text{cache}} = \max\left(CS^{\text{to}}, CS^{\text{from}}\right) \tag{12}$$

For MSRS, blocking is caused by *Save*, *Load* and *Restore* steps carried out by lower-priority tasks $k \in \text{lp}(i)$:

$$B_i^{\text{lp}} = \max_{k \in \text{lp}(i)} (CS^{\text{to}} + C_k^{\text{save}} + C_{k,1}^{\text{load}}, \max_{x \in [2,m]} C_{k,x}^{\text{load}}, C_k^{\text{restore}} + CS^{\text{from}}) \tag{13}$$

This fits into (10) as follows:

$$B_i^{\text{spm}} = \max(B_i^{\text{lp}}, C_i^{\text{restore}} + CS^{\text{from}}) \tag{14}$$

System Parameter	Upper Bound Value	
Clock Period	10	ns
Block Size	16	bytes
Local Memory Size	128	blocks
$\text{BRT}^{\text{cache}}$	310	ns
CS^{to}	9090	ns
CS^{from}	5500	ns
B^{cache}	9090	ns
C_i^{save}	$10S_i^{\text{spm}} + 480$	ns
BRT^{spm}	320	ns
$C_{i,j}^{\text{load}}$	$\text{BRT}^{\text{spm}} S_{i,j}^{\text{spm}} + 150$	ns
C_i^{restore}	$\text{BRT}^{\text{spm}} S_i^{\text{spm}} + 570$	ns

TABLE I
SYSTEM PARAMETERS OBTAINED FROM FPGA IMPLEMENTATION.

E. Discussion

The definitions given above make B_i^{spm} larger than B_i^{cache}, as existing critical sections (CS^{to}, CS^{from}) are extended. Furthermore, C_i^{spm} is typically similar to C_i^{cache}.

C_i^{spm} is only less than C_i^{cache} in special cases, e.g. if fewer blocks are loaded from external memory for SPM than cache because SPM allocation is able to pack the same code into fewer blocks by defragmenting, or because conflict misses occur for cache [20]. However, our benchmarks are chosen so that conflict misses do not occur (Section I), and the impact of defragmentation is unimportant within the tasks we used, only affecting the small minmax benchmark. C_i^{spm} may also be smaller than C_i^{cache} due to differences in block size or pipelining of transfers to SPM [10], but in this paper, we choose the same block size for both cache and SPM, and assume that SPM transfers are not pipelined.

Therefore, the success of MSRS depends entirely upon achieving some $\gamma_{i,j}^{\text{SRPD}}$ that is smaller than both $\gamma_{i,j}^{\text{ucbu}}$ and $\gamma_{i,j}^{\text{ecbu}}$. Put simply, the additional overhead of *Save* and *Restore* must be less than the overhead of CRPD. However, it is not possible to make such a comparison without considering the properties of task sets.

IV. MSRS IMPLEMENTATION

In this section, we determine equations for parameters such as $\text{BRT}^{\text{cache}}$ and C_i^{save} from a working system implementation, and then describe the task sets used for our experiments.

A. System Parameters

Table I shows the parameters of the hardware and system software, some of which are dependent on task sizes (S_j^{spm}) and region sizes ($S_{i,j}^{\text{spm}}$). We introduce BRT^{spm} as the time needed to load a single SPM block from external memory; notice that this is not part of C_i^{save} because only the address of each block needs to be *Saved*.

The parameters are based on the following hardware and system software components:

ARM7 CPU. We assume a timing-compositional architecture [19] which requires adopting a CPU that is free of timing anomalies. One of the best-known CPUs of this sort is the ARM7 CPU core which is supported by the Absint aiT WCET analysis tool [23]. The model of ARM7 used by aiT has configurable caches: the total size, block size and miss latency of each cache can be specified. Furthermore, aiT can be used to capture the ECB_i and UCB_i sets for each τ_i [15].

FPGA-based SPM. ARM7 does not include SPM. This is an issue because MSRS involves loading information from external memory to SPM. Without some model of this process, we have no way to find the relevant parameters. Therefore, we created an FPGA implementation of the system that included an SPM and hardware to load data into the SPM (Figure 2).

This FPGA implementation is not based on an ARM7 CPU, since ARM7 is not available to us in synthesizable form. Instead we use the similar Microblaze CPU, reasoning that the differences between Microblaze and ARM do not affect the performance of surrounding hardware such as external memory, cache and SPM. The same cache/SPM implementation can be used for both. The FPGA implementation allows us to capture the parameters of Microblaze+SPM by measurement; we then assume that these would be the same for a possible ARM7+SPM implementation if one existed.

Details of the FPGA implementation cannot be included in this paper due to space limitations; however, the design is downloadable from our web page[1]. Our measurements were performed using a Xilinx ML505 FPGA board, featuring a Virtex 5 LXT FPGA and external RAM, but the design uses a standard memory controller and CPU components and is adaptable to other recent Xilinx FPGA devices.

External RAM. We use a 256Mb SO-DIMM DDR2 module manufactured by Micron. The memory controller implements a periodic refresh as required for DDR2. This is activated every 7800ns. During the refresh, memory accesses are blocked for a few clock cycles. Our measurements were all captured *between* refreshes, on the grounds that there are more appropriate ways to deal with refreshes in a real-time context which are orthogonal to MSRS [24]. WCET, SRPD and CRPD analysis are not the right places to consider the possible impact of refreshes, which should be handled at the system level. Therefore, refreshes are not given further consideration in this paper - it is assumed throughout that the memory controller is always ready to respond to requests.

Real-time Operating System. Some parameters are a consequence of both the hardware and the operating system software; the context-switching time is an example. We adapted Carousel OS [10] to run on the FPGA hardware. The OS is resident in a system SPM (Figure 2). It is configurable at compile time to use either instruction cache or instruction SPM, which makes no difference to the scheduler or the device drivers, but changes how new tasks are invoked. For SPM, an implementation of MSRS is used, comprising the four steps *Save*, *Load*, *Execute*, *Restore* (Section III-A). For cache, only *Execute* is used.

B. Software Tasks

We used the same tasks as [16] as listed in Table II, excluding tasks that are larger than the local memory (i.e. $|\text{ECB}_i| >$

[1] http://www.cs.york.ac.uk/rts/rtslab/

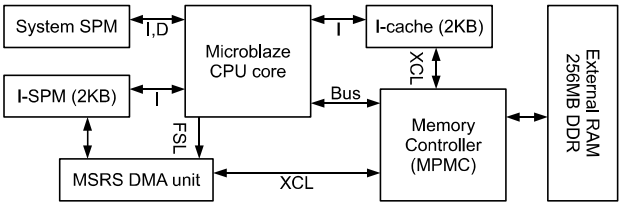

Fig. 2. Microblaze+SPM hardware implemented within an FPGA.

Task τ_i	C_i^{execute} (ns)	$\lvert\text{ECB}_i\rvert$	C_i^{cache} (ns)	$\lvert\text{UCB}_i\rvert$
binarysearch	2980	18	8560	13
bsort100	11074380	32	11084300	18
crc	2117220	73	2139850	61
fac	10460	13	14490	11
fibcall	13470	13	17500	7
fir	213630	55	230680	41
insertsort	61980	21	68490	11
matmult	8056310	52	8072430	20
minmax	2790	36	13950	12
ns	365910	33	376140	29
qsortexam	165440	87	192410	81
select	133750	77	157620	72

TABLE II
BASE TASKS FROM THE MRTC BENCHMARK COLLECTION [25].

128) in order to avoid conflict misses. Both cache and MSRS can be used with larger benchmarks [16], [10], but excluding conflict misses simplifies comparisons between MSRS and cache by (i) removing the effects that penalize cache (but not SPM) and are not related to preemption, and (ii) allowing the use of a simple region formation scheme for SPM.

The tasks are from the well-known MRTC benchmark collection [25]. The columns of Table II are:

1) Task τ_i name,
2) C_i^{execute}, determined by WCET analysis using aiT, with no instruction cache or data cache, so all memory accesses are zero-latency. An ARM7 CPU model [26] is assumed.
3) C_i^{cache} determined by:

$$C_i^{\text{cache}} = \text{BRT}^{\text{cache}}|\text{ECB}_i| + C_i^{\text{execute}} \tag{15}$$

Notice that (15) and (7) are very similar; the cache miss time $\text{BRT}^{\text{cache}}|\text{ECB}_i|$ is substituted for C_i^{load}.
4) $|\text{ECB}_i|$, the number of distinct cache blocks which might be used by the task as it executes [12], determined during WCET analysis.
5) $|\text{UCB}_i|$, the maximum number of *definitely-cached useful cache blocks* (DC-UCBs) in the task. DC-UCBs are cached blocks that are reused during execution [15].

C. Software Task Sets

The task set $\{\tau_1, \tau_2, ..., \tau_n\}$ is chosen at random from Table II, i.e. each τ_i has an equal probability of having the properties of any one of the rows of Table II.

We assign a utilization U_i to each task τ_i in order to match a total utilization $U = \sum_i U_i$. This assignment is performed using the UUnifast algorithm [27]. As WCETs are already defined, utilizations are assigned by setting $T_i = \frac{1}{U_i} C_i^{\text{cache}}$.

We assume that deadline is equal to period for all tasks ($D_i = T_i$) and use deadline monotonic priority assignment, so that tasks with shorter deadlines always have higher priorities. Deadline monotonic priority assignment is optimal if preemption and context switch costs are negligible.

Like [16], we pick a random starting point within the cache. The ECB set for task τ_1 begins at this point and occupies $|\text{ECB}_1|$ contiguous blocks. The ECB sets for subsequent tasks are arranged consecutively, i.e. ECB_{i+1} begins immediately after ECB_i. The UCB sets are generated as subsets of the ECB sets, and again from [16], we choose a random starting point within ECB_i for each task τ_i. The UCB set for task τ_i begins at this point, and occupies $|\text{UCB}_i|$ contiguous blocks.

This completes the model of each task, and there is enough information to determine if the task set is schedulable if executed with cache. We use (6) for the schedulability test, i.e. the task set is schedulable if $\forall i. R_i^{\text{CRPD}} \leq D_i$.

By generating many task sets with a specific utilization U, and checking for schedulability, we can find the success ratio: the proportion of task sets that are schedulable at that utilization, and thus compare MSRS and cache.

D. Good Conditions, Poor Conditions

Figure 1 illustrates the comparison approach used in this section. The same task sets are analyzed with both cache and MSRS as a way to compare their schedulability.

When combined with task set parameters such as T_i, Table II gives all the necessary information for schedulability analysis with CRPD-RTA. However, as S_i^{spm} is absent, there is not enough information for analysis with SRPD-RTA. Substituting parameters from Table I into (8) leaves S_i^{spm} as an unknown:

$$\begin{aligned}\gamma_{i,j}^{\text{SRPD}} &= C_j^{\text{save}} + C_j^{\text{restore}} \\ &= 330 S_j^{\text{spm}} + 1050\end{aligned} \tag{16}$$

The largest likely $S_i^{\text{spm}} = |\text{ECB}_i|$. This occurs when there is only one region. The smallest likely $S_i^{\text{spm}} = |\text{UCB}_i|$ because regions are typically formed where code is reused, so that regions typically contain an entire loop, and are thus the same size as $|\text{UCB}_i|$. These observations give upper and lower bounds on $\gamma_{i,j}^{\text{SRPD}}$. Similar substitutions into (14) can be made to determine B_i^{spm}:

$$\begin{aligned}B_i^{\text{spm}} = \max(&\max_{k \in \text{lp}(i)} (330 S_k^{\text{spm}} + 9720), \\ &320 S_i^{\text{spm}} + 6070)\end{aligned} \tag{17}$$

The total number of blocks loaded is $|\text{ECB}_i|$, assuming that the code size is unchanged by region formation, so C_i^{spm} is:

$$C_i^{\text{spm}} = 320|\text{ECB}_i| + 150 + C_i^{\text{execute}} \tag{18}$$

To carry out a comparison, we tested task sets with combined utilization $U \in [0, 1]$. For each value of U, 100000 task sets were generated, each containing exactly $n = 15$ tasks. Each task was selected at random from Table II.

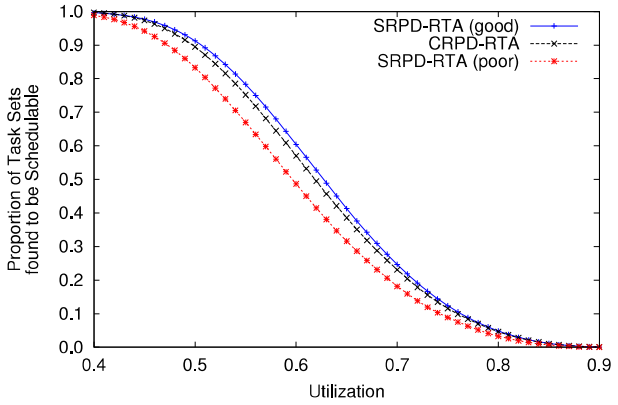

Fig. 3. The proportion of task sets of size $n = 15$ with utilization $U \in [0.4, 0.9]$ that are schedulable using cache with CRPD-RTA analysis, compared with MSRS in "good" conditions ($S_i^{\text{spm}} = |\text{UCB}_i|$) and MSRS in "poor" conditions ($S_i^{\text{spm}} = |\text{ECB}_i|$), each with SRPD-RTA analysis.

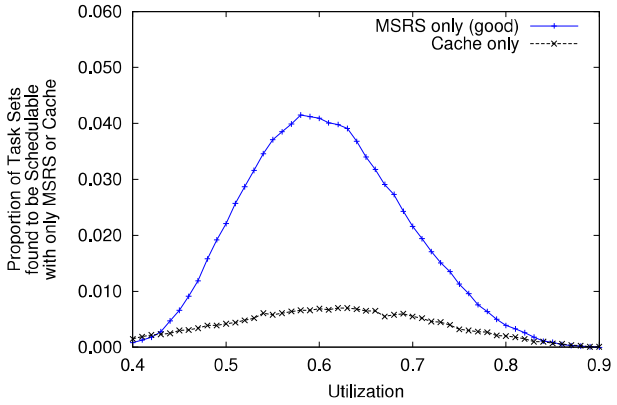

Fig. 4. The proportion of task sets from Figure 3 that were deemed schedulable by CRPD-RTA but not SRPD-RTA ("Cache only") and the number that were deemed schedulable by SRPD-RTA but not CRPD-RTA ("MSRS only (good)").

The task sets were tested in three scenarios: (i) cache, (ii) MSRS in "good" conditions (where $S_i^{\text{spm}} = |\text{UCB}_i|$), and (iii) MSRS in "poor" conditions (where $S_i^{\text{spm}} = |\text{ECB}_i|$) For scenarios (ii) and (iii), $\gamma_{i,j}^{\text{SRPD}}$ is defined by (16), B_i^{spm} is defined by (17) and C_i^{spm} is defined by (18).

Figure 3 shows the results. Cache is clearly preferable to MSRS in "poor" conditions; far more task sets are schedulable with cache. However, in "good" conditions, MSRS and cache allow a similar number of task sets to be scheduled.

Figure 4 shows an alternative view of Figure 3, comparing MSRS in "good" conditions to cache. This plot shows the number of task sets that were deemed schedulable by CRPD-RTA but not SRPD-RTA ("Cache only") and the number that were deemed schedulable by SRPD-RTA but not CRPD-RTA ("MSRS only"). This tells us that while MSRS and cache allow a similar number of task sets to be scheduled (Figure 3), there are important differences between the two approaches. As the MSRS only curve shows, some task sets are only schedulable with MSRS. A smaller number are only schedulable with cache. Therefore, SRPD and CRPD are *incomparable* in that neither dominates the other across all possible task sets. We explore this difference in section V.

E. Realistic Conditions

We now look at what happens when S_i^{spm} and C_i^{spm} are given realistic values from our hardware implementation. This requires splitting each task into two or more regions; if we retain only one region, we have $S_i^{\text{spm}} = |\text{ECB}_i|$, and results are poor (Figure 3).

Splitting tasks into regions for SPM allocation is a whole topic in itself [5]. In this paper, we choose a simple scheme which is effective for small tasks.

Consider the binarysearch benchmark. This has the following structure when compiled for ARM and executed:

1) Run initialization code (68 bytes),
2) Run binary search loop (208 bytes),
3) Run completion code, return (12 bytes).

Only the looped code is ever reused, and hence $|\text{UCB}_i| = \lceil \frac{208}{16} \rceil = 13$ blocks. However, all of the code is executed at least once, so $|\text{ECB}_i| = \lceil \frac{68+208+12}{16} \rceil = 18$ blocks. If region boundaries are placed at each transition between sequential code and a loop, then each of the three parts becomes a separate region. The space used for the loop code can be shared with the space used for initialization, so the program is executed as follows:

1) Load initialization code (68 bytes plus 16 bytes to load next region; $\lceil \frac{68+16}{16} \rceil = 6$ blocks),
2) Run initialization code,
3) Load binary search loop into SPM (208 bytes plus 16 bytes to load next region; $\lceil \frac{208+16}{16} \rceil = 14$ blocks),
4) Run binary search loop,
5) Load completion code into SPM (12 bytes, 1 block),
6) Run completion code and return.

If τ_i = binarysearch, then $S_{i,1}^{\text{spm}} = 6$, $S_{i,2}^{\text{spm}} = 14$, $S_{i,3}^{\text{spm}} = 1$, and from (11), $S_i^{\text{spm}} = \max\{6, 14, 1\} = 14$.

The loading costs for these regions are $C_{i,1}^{\text{load}} = 2070$, $C_{i,2}^{\text{load}} = 4630$, $C_{i,3}^{\text{load}} = 470$. Substituting these into (7) with C_i^{execute} from Table II gives $C_i^{\text{spm}} = 10150$ns.

F. Region Formation

Our experimental software (downloadable; see footnote 1) applies the process described for binarysearch to divide the other tasks in Table II into multiple regions. In each case, the software forms a linear sequence of m regions where region $x + 1$ always follows region x. In most cases, $m = 3$: the regions being initialization, execution, and completion. In some cases, more regions are possible, e.g. matmult has five regions because the initialization phase involves a loop, and hence some blocks are reused.

All tasks in Table II form at least $m = 3$ regions; some (bsort100, matmult, sqrt) form $m = 5$ regions. The region formation process is further optimized by merging adjacent pairs of regions whenever this does not increase S_i^{spm}, reducing m.

Task τ_i	C_i^{spm} (ns)	S_i^{spm}	$\frac{C_i^{\text{spm}}}{C_i^{\text{cache}}}$	$\frac{S_i^{\text{spm}}}{\lvert\text{ECB}_i\rvert}$	$\frac{S_i^{\text{spm}}}{\lvert\text{UCB}_i\rvert}$
binarysearch	10150	14	1.186	0.778	1.077
bsort100	11085710	19	1.000	0.594	1.056
crc	2141990	61	1.001	0.836	1.000
fac	15710	10	1.084	0.769	0.909
fibcall	18720	6	1.070	0.462	0.857
fir	232320	41	1.007	0.745	1.000
insertsort	69790	11	1.019	0.524	1.000
matmult	8074660	21	1.000	0.404	1.050
minmax	11240	11	0.806	0.306	0.917
ns	377090	29	1.003	0.879	1.000
qsortexam	194370	82	1.010	0.943	1.012
select	160120	72	1.016	0.935	1.000

TABLE III
TASKS AFTER REGION FORMATION FOR SPM.

This simple region formation process is inevitably restricted because it does not allow regions to exist as proper subsets of loops, and it does not allow a conditional statement to choose between more than one region. Better region assignments may exist. Finding high-quality or optimal SPM allocations for tasks is the subject of ongoing work.

G. Software Tasks for SPM

Region formation gives each task the parameters shown in Table III. The columns of Table III are:

1) Task name (as in Table II).
2) C_i^{spm}, determined using (7).
3) S_i^{spm}, determined using (11).
4) The ratio $\frac{C_i^{\text{spm}}}{C_i^{\text{cache}}}$, showing the difference in WCET from SPM to cache. The WCET for SPM is typically larger because the overheads in (7) are greater than (15).
5) The ratio $\frac{S_i^{\text{spm}}}{\lvert\text{ECB}_i\rvert}$, showing the difference in the local memory requirement for each task. We see that forming regions has greatly reduced the space requirement.
6) The ratio $\frac{S_i^{\text{spm}}}{\lvert\text{UCB}_i\rvert}$, showing that SPM allocation has got close to (and sometimes exceeded) the "good" condition of Figure 3.

Figure 5 shows a comparison of cache and MSRS in realistic conditions. The graph is a copy of Figure 3 with new data added for SRPD-RTA (real) generated using the real task set parameters in Tables II and III. Again, 100000 task sets of size $n = 15$ were tested at each U. The difference between SRPD-RTA (good) and SRPD-RTA (real) is almost indistinguishable, though there are differences: for instance, in Table III, S_i^{spm} is sometimes larger than $\lvert\text{UCB}_i\rvert$.

V. INVESTIGATION

This paper has shown that in practical implementations, MSRS and cache can have very similar performance. Nevertheless, there are differences. The goal of this section is to explain the causes of these differences.

In order to do this, we use the comparison approach illustrated in Figure 1. We begin with a baseline comparison of SRPD-RTA and CRPD-RTA. Section IV-D's "good conditions" turned out to be similar to realistic conditions (Figures 3 and 5), so we use Figure 3 as the baseline with $S_i^{\text{spm}} = \lvert\text{UCB}_i\rvert$. $\gamma_{i,j}^{\text{SRPD}}$, B_i^{spm} and C_i^{spm} are defined by (16), (17) and (18).

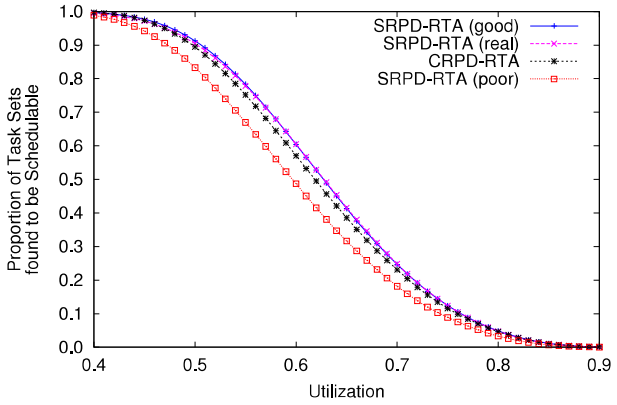

Fig. 5. The proportion of task sets of size $n = 15$ with utilization $U \in [0.4, 0.9]$ that are schedulable using MSRS in "good", "real" and "poor" conditions with SRPD-RTA analysis, and using cache with CRPD-RTA analysis. The graph shows that SRPD-RTA (good) is very similar to SRPD-RTA (real).

A. Number of Tasks

The success of MSRS is highly dependent on the number of tasks n. The difference between CRPD-RTA and SRPD-RTA for different n may be illustrated using a *weighted schedulability measure* $W_y(p)$ [28]. The measure combines data for all task sets generated with a particular parameter set p and tested with a particular schedulability test y into a single value, namely a weighted integration of the area under the curve in a graph such as Figure 3. The value $W_y(p)$ can then be plotted on a graph for various p. The weighted schedulability measure is defined as follows:

$$W_y(p) = \frac{\sum_\tau u(\tau) S_y(\tau, p)}{\sum_\tau u(\tau)} \tag{19}$$

where $u(\tau)$ is the utilization of task set τ and $S_y(\tau, p)$ is 1 if task set τ is schedulable according to test y and parameter set p, otherwise 0. In Figure 3, $W_{\text{CRPD-RTA}} = 0.395$ and $W_{\text{SRPD-RTA}} = 0.404$ in "good" conditions. In realistic conditions (Section IV-G), $W_{\text{SRPD-RTA}} = 0.403$.

Let p be the number of tasks, n. Figure 6 shows what happens when we vary n from 1 to 30, picking each task at random from Table II, and using idealized parameters as for Figure 3. (The baseline is $n = 15$ and 10000 task sets were generated for each U.)

The weighted schedulability measure is the same for a single task, because of the idealized assumptions. For small numbers of tasks, cache is preferable: $W_{\text{CRPD-RTA}}(n) > W_{\text{SRPD-RTA}}(n)$. But for larger numbers of tasks ($n > 10$), SPM is advantageous. We explain why this is in Section V-F.

B. Load time ratio

In our implementation, the cost of loading a block from SPM is $\text{BRT}^{\text{spm}} = 320$ns; the cost of loading a block from cache is $\text{BRT}^{\text{cache}} = 310$ns (Table I). The difference is the need

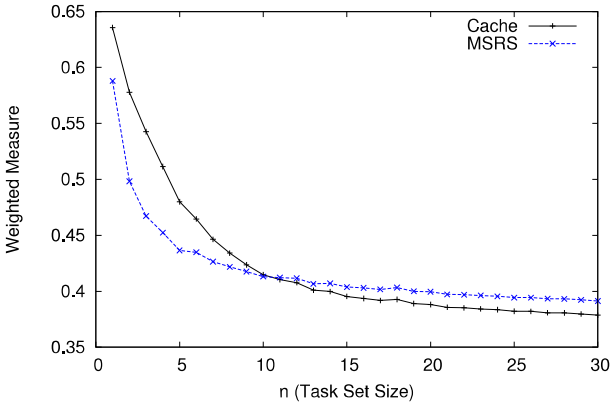

Fig. 6. Weighted measure of schedulability for MSRS and cache as the task set size is changed. Other parameters as in Section IV-D.

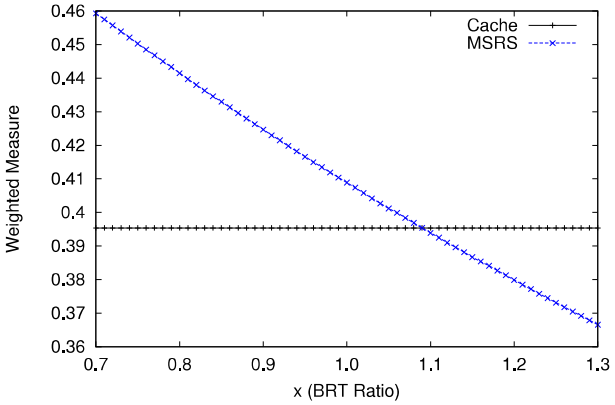

Fig. 7. Weighted measure of schedulability for MSRS and cache with $\text{BRT}^{\text{spm}} = x\text{BRT}^{\text{cache}}$. Other parameters as in Section IV-D.

to explicitly order the transfer, which requires the execution of an additional instruction. For this comparison, we define $\text{BRT}^{\text{spm}} = x\text{BRT}^{\text{cache}}$. The baseline $x = 1.03$.

$x < 1$ is possible if the block size for SPM is larger than the block size for cache, because larger blocks can be transferred more efficiently across a bus. Therefore, we explore the range $x \in [0.7, 1.3]$.

Figure 7 shows the result, assuming $n = 15$ tasks. It is clear that schedulability is very sensitive to BRT^{spm} and $\text{BRT}^{\text{cache}}$.

If $x = 1$ so $\text{BRT}^{\text{spm}} = \text{BRT}^{\text{cache}}$, $W_{\text{SRPD-RTA}} = 0.409$, which is much better than the $W_{\text{CRPD-RTA}}$ equivalent (0.395). However, with $x = 1.03$, we have $W_{\text{SRPD-RTA}} = 0.404$: a very small increase in x has greatly reduced schedulability. $x = 1.1$ would result in $W_{\text{SRPD-RTA}} = 0.394$, worse than $W_{\text{CRPD-RTA}}$.

BRT^{spm} affects both $\gamma_{i,j}^{\text{SRPD}}$ and C_i^{spm}, so it is unsurprising that its impact is so severe. Minimizing this cost is important.

C. S^{spm}, $|ECB|$ ratio

Figure 8 shows the effect of changing the relationship between S_i^{spm}, $|\text{UCB}_i|$ and $|\text{ECB}_i|$. On the x-axis, a value of $x = 0$ corresponds to $S_i^{\text{spm}} = |\text{UCB}_i|$, and a value of $x = 1$ corresponds to $S_i^{\text{spm}} = |\text{ECB}_i|$, i.e.

$$S_i^{\text{spm}} = |\text{UCB}_i| + (|\text{ECB}_i| - |\text{UCB}_i|)x \tag{20}$$

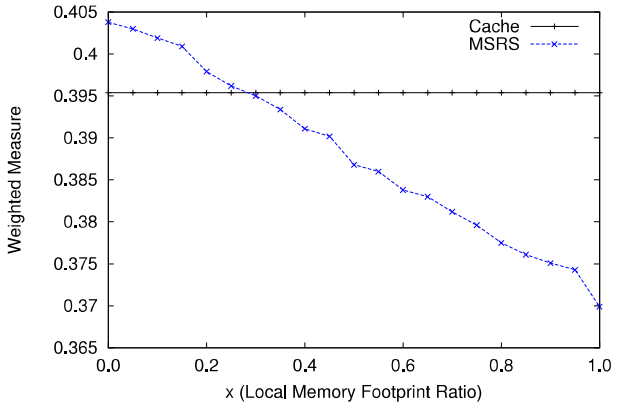

Fig. 8. Weighted measure of schedulability for MSRS and cache for $S_i^{\text{spm}} = |\text{UCB}_i| + (|\text{ECB}_i| - |\text{UCB}_i|)x$. Other parameters as in Section IV-D.

Figure 8 indicates that the success of MSRS depends on the ability of the SPM-based technology to reduce the local memory footprint of each task. S_i^{spm} must be less than $|\text{ECB}_i|$ if MSRS is to improve on cache.

D. Local Memory Size

Figure 9 shows the effect of changing the local memory size. For this graph, the local memory size is calculated as 2^x, where x is the value on the X axis. For instance, $x = 7$ corresponds to 128 blocks, which is the baseline local memory size (Table I).

Because we have set out to ignore the effects of conflict misses, this experiment involves ignoring the tasks in Table II that require more than 2^x blocks (i.e. where $|\text{ECB}_i| > 2^x$). Therefore, the generated task sets for $x < 7$ are different to those for all other x. Furthermore, we are unable to test $x < 5$, because our benchmark tasks are too large ($2^5 = 32$ blocks).

We see that MSRS is preferable to cache when the local memory size is small. Additionally, the performance of MSRS is independent of the local memory size (in Figure 9, the performance of MSRS only varies when the set of valid tasks changes). Whereas the performance of cache is dependent upon the local memory size, up to the point where the cache is large enough that the ECB sets are disjoint and no task evicts blocks belonging to another. There is also a size (approximately $2^8 = 256$ blocks) where cache becomes preferable because the impact of CRPD has become less than the impact of SRPD.

E. The Impact of Blocking

Earlier comparisons use the baseline definitions of B_i^{spm} (17) and B_i^{cache} (12). With these definitions, $B_i^{\text{spm}} > B_i^{\text{cache}}$. The critical sections invoked as tasks begin and end have greater WCETs for MSRS.

Another implementation might make *Save*, *Load* and *Restore* interruptible, almost reducing B_i^{spm} to B_i^{cache}. Tasks

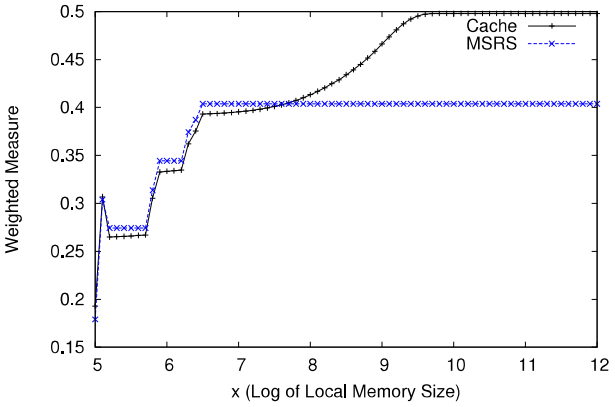

Fig. 9. Weighted measure of schedulability for MSRS and cache for local memory size 2^x blocks. Other parameters as in Section IV-D.

would still be able to block themselves, so the minimum B_i^{spm} is similar to (12):

$$B_i^{\text{spm}} = \max\left(CS^{\text{to}}, C_i^{\text{restore}} + CS^{\text{from}}\right) \tag{21}$$

The effect of substituting (21) for (17) is a negligible increase in the number of task sets schedulable with MSRS: $W_{\text{SRPD-RTA}}$ improves from 0.4035 to 0.4038. This is much less than the significance of reducing BRT^{spm} or S^{spm}. Therefore, if we are forced to choose between reducing blocking and reducing BRT^{spm} while reimplementing MSRS, we should certainly choose the latter.

F. General Observations

Figures 6 and 9 illustrate essentially the same behavior, namely the effect of contention for local memory space, which affects cache and MSRS in different ways. Tasks are more likely to evict blocks belonging to other tasks when the task set size is increased (Figure 6) and when the local memory size is reduced (Figure 9).

Increasing the task set size means that $\gamma_{i,j}^{\text{ucbu}}$ and $\gamma_{i,j}^{\text{ecbu}}$ (see (4) and (5)) become closer to $\text{BRT}^{\text{cache}}|\text{ECB}_j|$, their upper bound. This is because more tasks means there are more ways that a low-priority task τ_i could be preempted by some high-priority task τ_j; not just directly, but also indirectly during preemption by some other task τ_k where $k \in \text{aff}(i,j)$. In other words, CRPD tends towards the assumption that all of task τ_i is evicted from cache by preemption. SRPD does not do this; $\gamma_{i,j}^{\text{SRPD}}$ is dependent only on the properties of task τ_j and not on the set of preempted tasks (8).

Similarly, reducing the local memory size causes $\gamma_{i,j}^{\text{ucbu}}$ and $\gamma_{i,j}^{\text{ecbu}}$ to become closer to $\text{BRT}^{\text{cache}}|\text{ECB}_j|$ because fewer blocks are available in cache, and hence ECB_j and UCB_j grow closer.

There is a tradeoff between the number of tasks and the local memory size. Large numbers of tasks may be schedulable with cache given sufficient local memory. If the local memory size is insufficient for the number of tasks, MSRS may be preferable, because it is unaffected by the number of tasks.

The phenomenon observed here is partly due to the imprecision and overestimation of CRPD-RTA analysis, which increases with the number of tasks.

Figure 7 illustrates a related effect. It may be that $\gamma_{i,j}^{\text{SRPD}}$ is smaller than $\gamma_{i,j}^{\text{ucbu}}$ and $\gamma_{i,j}^{\text{ecbu}}$ if $\text{BRT}^{\text{spm}} = \text{BRT}^{\text{cache}}$, this improvement will be lost if $\text{BRT}^{\text{spm}} > \text{BRT}^{\text{cache}}$. It is therefore crucial that BRT^{spm} is as close to $\text{BRT}^{\text{cache}}$ as possible. Unfortunately, even a small overhead (320ns versus 310ns) has a significant impact. Improved implementations of MSRS could address this.

It is already noted that Figure 8 shows the importance of minimizing S_i^{spm}, because the success of MSRS depends on this. However, Figure 8 also provides a clue about how to improve the schedulability using cache. If tasks using cache can be arranged such that $\text{UCB}_i = \text{ECB}_i$, the task set will approach the schedulability available from MSRS. This could be achieved through a region-forming process (Section IV-F).

VI. Related Work

Schedulability comparisons are a common way to evaluate analyses for real-time systems [27] and earlier work on CRPD also made use of them [15], [16]. Generated task sets may be entirely synthetic, but for this paper we require realistic information about the relationship between parameters such as S_i^{spm} and $|\text{UCB}_i|$, so it is essential to base the task sets on real tasks taken from benchmark software [25]. We used the same benchmarks as earlier work on MSRS [10] and earlier work on CRPD [16].

SPM has been used within multitasking real-time systems in previous work. MSRS may be regarded as an improvement of earlier approaches to prevent one task evicting local memory resources used by another, such as static cache locking [29] and static SPM partitioning [9], [7]. Simpler forms of MSRS have been proposed [30], [9], but the one used in this paper has the important distinction of enabling any task to use an arbitrary number of blocks in SPM independently of all other tasks [10].

MSRS may only be used if task preemption is properly nested. Tasks must preempt and complete in stack order, so that the *Save* and *Restore* steps work correctly. This is suitable for real-time systems that use the Stack Resource Policy (SRP) [31] for resource access, but not for arrangements where multiple tasks execute simultaneously, as observed with *precision-timed architecture* (PRET) CPUs [7]. PRET uses a time-predictable form of *simultaneous multithreading* (SMT) in which all tasks may access SPM at the same time, requiring static partitioning of SPM.

VII. Conclusion

In this paper, we have presented a multitasking scratchpad reuse scheme (MSRS) for the dynamic partitioning of SPM space between tasks. This is accompanied by the notion of *scratchpad-related preemption delay* (SRPD) which, being an analog of *cache-related preemption delay* (CRPD), indicates the cost imposed upon a preempted task by preemption in a multitasking system.

We have compared MSRS with its cache equivalent through comparisons of SRPD and CRPD within both realistic and idealistic experiments. The comparisons were carried out by applying *worst-case response time* (WCRT) analysis with SRPD and CRPD (Figure 1). We have shown that MSRS is preferable to cache for certain task sets, in the sense that some task sets that are not schedulable with cache are schedulable with MSRS. A large task set with a small local memory is more likely to be schedulable with MSRS. Unlike CRPD, SRPD is independent of the number of tasks and the local memory size.

Lastly, we have determined that it is very important to minimize the overheads of using SPM, particularly the block reload time BRT^{spm}, which is larger than the cache equivalent within our experimental system implementation. This motivates improved hardware support for MSRS.

Future work may relax the simplifying assumptions made for this paper by considering set-associative caches, data caches and more complex tasks. If individual tasks are larger than the local memory size, then the comparisons may become more complex as conflict misses occur and advanced SPM allocation techniques may be required to further subdivide regions. These will increase C_i^{cache} and C_i^{spm} for each task, and may reduce S_i^{spm}. Simulations could also be used to show how much of the difference between cache and MSRS is caused by imprecision in the analyses and how much is present in a real system.

ACKNOWLEDGMENTS

This work was supported by EPSRC project TEMPO, number EP/G055548/1 and FP7 project T-CREST, number 288008.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[2] J. Whitham and N. Audsley, "Investigating average versus worst-case timing behavior of data caches and data scratchpads," in *Proc. ECRTS*, 2010, pp. 165–174.

[3] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. DATE*. Washington, DC, USA: IEEE Computer Society, 2002, p. 409.

[4] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. R&D*, vol. 49, no. 4/5, pp. 589–604, 2005.

[5] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *Proc. DATE*, 2007, pp. 1484–1489.

[6] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET Centric Data Allocation to Scratchpad Memory," in *Proc. RTSS*, 2005, pp. 223–232.

[7] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *Proc. CASES*, 2008, pp. 137–146.

[8] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer, "A dynamic instruction scratchpad memory for embedded processors managed by hardware," in *Proc. ARCS*, 2011, pp. 122–134.

[9] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel, "Scratchpad sharing strategies for multiprocess embedded systems: A first approach," in *Proc. ESTImedia*, 2005, pp. 115–120.

[10] J. Whitham and N. Audsley, "Explicit Reservation of Local Memory in a Predictable, Preemptive Multitasking Real-time System," in *Proc. RTAS*, 2012.

[11] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Trans. Comput.*, vol. 47, no. 6, pp. 700–713, Jun. 1998.

[12] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *Proc. RTAS*, 1996, pp. 204–212.

[13] Y. Tan and V. Mooney, "Timing analysis for preemptive multitasking real-time systems with caches," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 1, Feb. 2007.

[14] J. Staschulat, S. Schliecker, and R. Ernst, "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay," in *Proc. ECRTS*, 2005, pp. 41–48.

[15] S. Altmeyer and C. Maiza, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, pp. 707–719, 2010.

[16] S. Altmeyer, R. Davis, and C. Maiza, "Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," in *Proc. RTSS*, 2011, pp. 261–271.

[17] ——, "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," *Real-Time Systems*, vol. 48, pp. 499–526, 2012. [Online]. Available: http://dx.doi.org/10.1007/s11241-012-9152-2

[18] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[19] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, 2009.

[20] M. D. Hill and A. J. Smith, "Evaluating associativity in cpu caches," *IEEE Trans. Comput.*, vol. 38, no. 12, pp. 1612–1630, Dec. 1989.

[21] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (can) schedulability analysis: Refuted, revisited and revised," *Real-Time Syst.*, vol. 35, no. 3, pp. 239–272, Apr. 2007.

[22] R. I. Davis, A. Zabos, and A. Burns, "Efficient exact schedulability tests for fixed priority real-time systems," *IEEE Trans. Comput.*, vol. 57, no. 9, pp. 1261–1276, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1109/TC.2008.66

[23] AbsInt Angewandte Informatik GmbH, "aiT Worst-Case Execution Time Analyzers," http://www.absint.com/ait/, 2012.

[24] B. Bhat and F. Mueller, "Making dram refresh predictable," *Real-Time Syst.*, vol. 47, no. 5, pp. 430–453, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1007/s11241-011-9129-6

[25] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks - Past, Present and Future," in *Proc. WCET*, July 2010. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=2284

[26] D. Seal, *ARM Architecture Reference Manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[27] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, no. 1-2, pp. 129–154, May 2005.

[28] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability," in *Proc. OSPERT*, 2010, pp. 33–44.

[29] X. Vera, B. Lisper, and J. Xue, "Data cache locking for tight timing calculations," *Trans. on Embedded Computing Sys.*, vol. 7, no. 1, pp. 1–38, 2007.

[30] H. Takase, H. Tomiyama, and H. Takada, "Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems," in *Proc. DATE*, 2010, pp. 1124–1129.

[31] T. P. Baker, "Stack-based scheduling of real-time processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–100, 1991.